Demonstration of Buffer Overflow and TCP Port Scanning in C

Author: Danmu Agu

---

1. Introduction

This report presents a cybersecurity project developed using the C programming language, focusing on two primary areas of system security: buffer overflow vulnerabilities and TCP port scanning. These components were chosen to demonstrate core competencies in memory management, secure coding, and network communications at the system level.

2. Objectives

1. To demonstrate how buffer overflow vulnerabilities occur and how they can be mitigated.
2. To build a simple TCP port scanner capable of identifying open ports within a specified range.
3. To gain practical experience with C system programming, socket programming, and security best practices.

---

3. Project Components

Buffer Overflow Demonstration

Two C source files were developed to illustrate this concept:

1. **vuln.c**: Contains a function vulnerable to buffer overflow by using the unsafe `gets()` function.
2. **safe.c**: Implements a secure version using `fgets()` with proper bounds checking.

Functionality:

1. Demonstrates how malicious input can exploit a buffer overflow to redirect control flow.
2. Includes a `win()` function which is unreachable under normal execution but can be triggered via overflow.
3. Designed for use with `gdb` to inspect stack memory and determine function addresses.

TCP Port Scanner

A functional TCP port scanner was implemented in **scanner.c**. It performs the following:

1. Accepts an IPv4 address and a port range as command-line arguments.
2. Attempts TCP connections to each port in the specified range.
3. Identifies and prints open ports.
4. Uses `setsockopt()` to apply a 1-second timeout to avoid long blocking on closed ports.
5. Validates input arguments and handles edge cases.

---

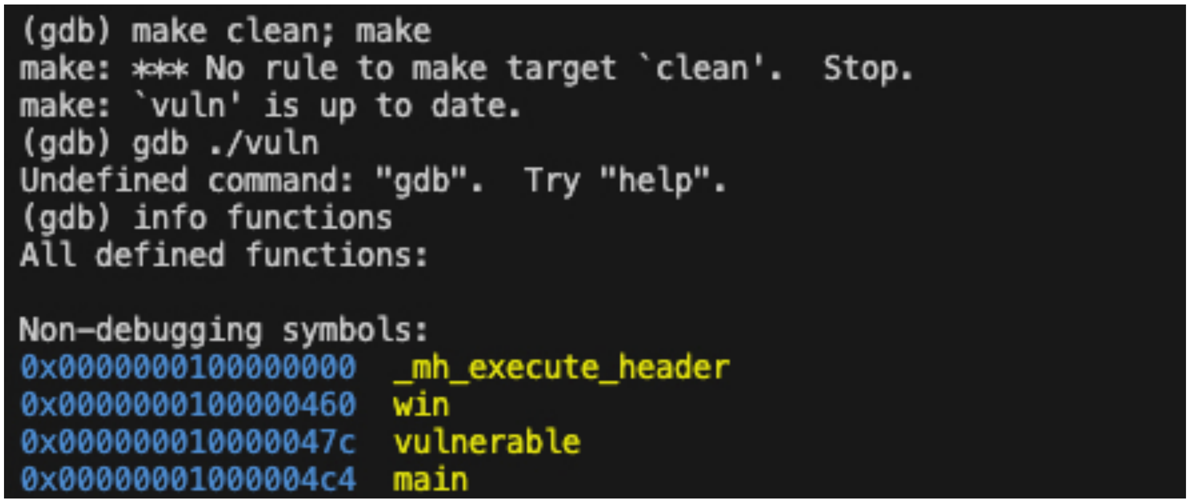
4. Technical Features

Feature	Description
Buffer Overflow	Unsafe memory access via stack-based buffer
Secure Coding	Use of `fgets()`, input bounds checking
Port Scanning	Socket programming with connection attempts
Timeout Handling	`setsockopt()` with `SO_RCVTIMEO` and `SO_SNDTIMEO`
Argument Validation	Range checks on ports, IP format validation

---

5. Demonstration and Screenshots

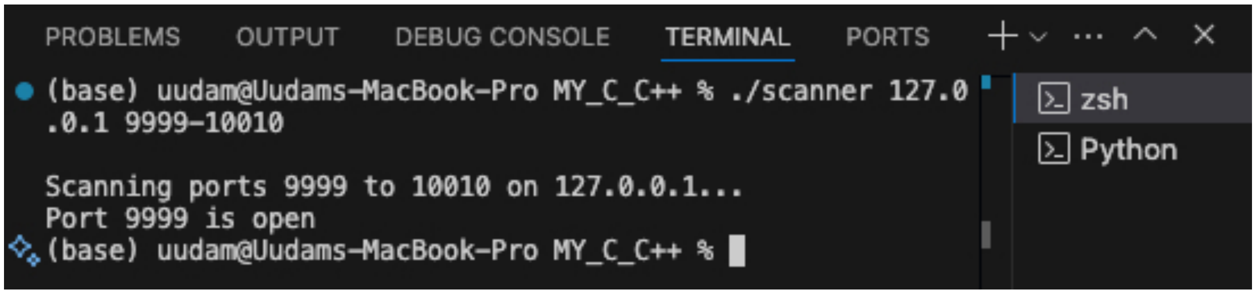
Buffer Overflow via gdb: Snapshot of stack memory manipulation.



```
(gdb) make clean; make
make: *** No rule to make target `clean'.  Stop.
make: `vuln' is up to date.
(gdb) gdb ./vuln
Undefined command: "gdb".  Try "help".
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000100000000  _mh_execute_header
0x0000000100000460  win
0x000000010000047c  vulnerable
0x00000001000004c4  main
```

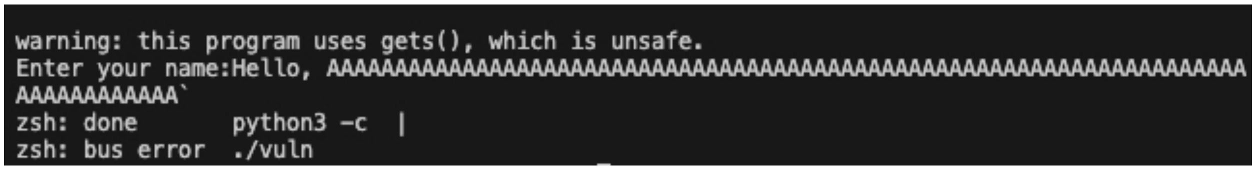
Successful Port Scan: Terminal output confirming port 9999 is open on localhost.



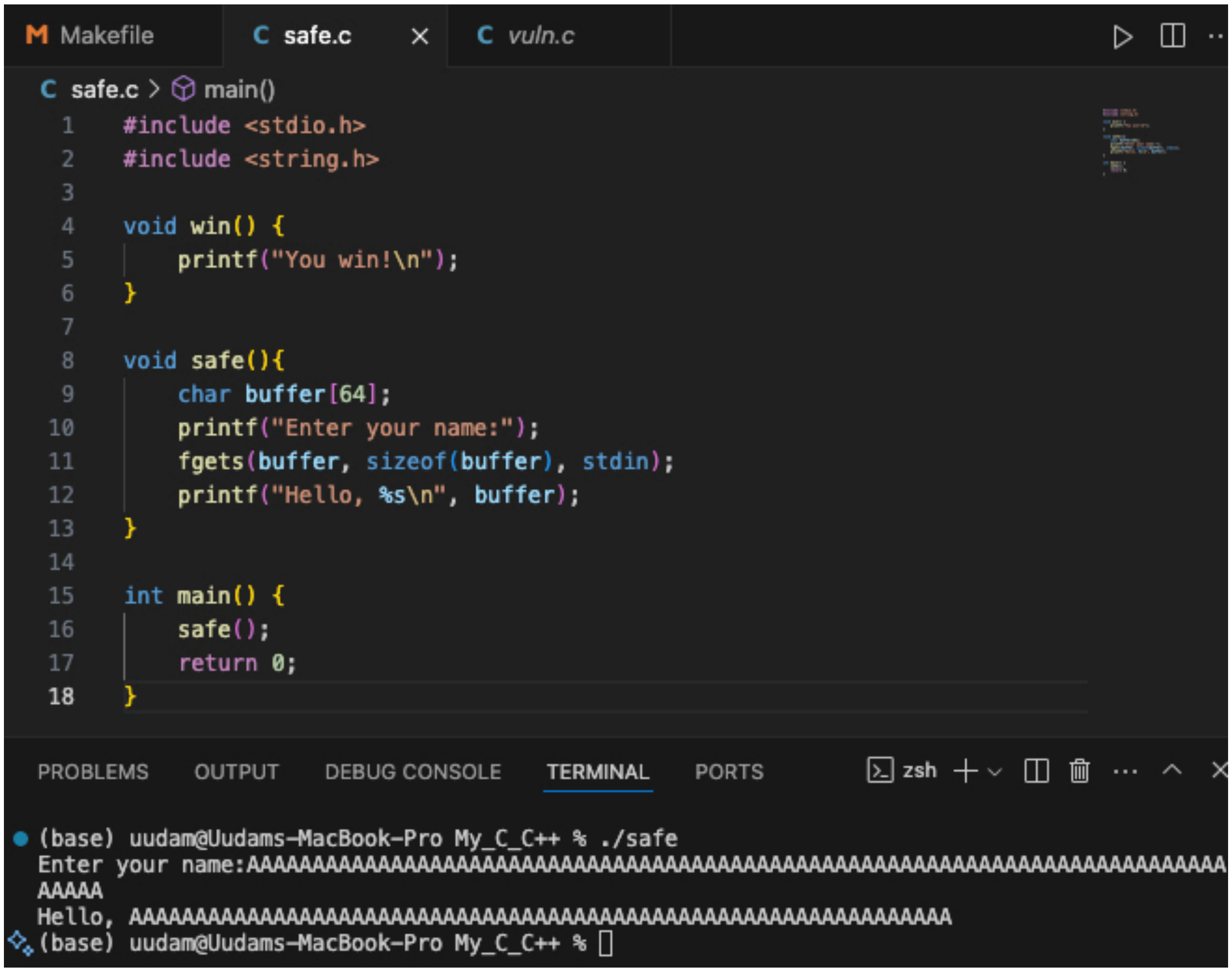
```
(base) uudam@Uudams-MacBook-Pro MY_C_C++ % ./scanner 127.0
.0.1 9999-10010

Scanning ports 9999 to 10010 on 127.0.0.1...
Port 9999 is open
(base) uudam@Uudams-MacBook-Pro MY_C_C++ %
```

Safe vs. Unsafe Behavior: Screenshots showing contrast between vulnerable and secure versions.



```
warning: this program uses gets(), which is unsafe.
Enter your name:Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA`
zsh: done       python3 -c  |
zsh: bus error  ./vuln
```



```c
#include <stdio.h>
#include <string.h>

void win() {
    printf("You win!\n");
}

void safe(){
    char buffer[64];
    printf("Enter your name:");
    fgets(buffer, sizeof(buffer), stdin);
    printf("Hello, %s\n", buffer);
}

int main() {
    safe();
    return 0;
}
```

```
(base) uudam@Uudams-MacBook-Pro My_C_C++ % ./safe
Enter your name:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(base) uudam@Uudams-MacBook-Pro My_C_C++ %
```

6. Usage Instructions

Compilation

Using the provided `Makefile`:

```
make vuln      # Builds vulnerable example
make safe      # Builds safe example
make scanner   # Builds port scanner
```

Running the Scanner

1. Host a local test server:

```
python3 -m http.server 9999 --bind 127.0.0.1
```

2. Execute scanner:

```
./scanner 127.0.0.1 9990-10010
```

Expected Output:

```
Port 9999 is open
```

---

7. Directory Structure

```
BufferOverflow_PortScanner/
├── vuln.c              // Vulnerable buffer overflow code
├── safe.c              // Secure version
├── scanner.c           // TCP port scanner
├── Makefile            // Build script
├── screenshots/        // Project demonstration images
├── .gitignore          // Ignore system/binary files
└── Project_Report.pdf
```

---

8. Learning Outcomes

Through the development and analysis of this cybersecurity project, I gained a deep understanding of buffer overflow vulnerabilities, learned to identify and exploit stack overflow vulnerabilities using the unsafe gets() function, and mastered secure coding practices to mitigate such risks by implementing fgets() with proper bounds checking. I acquired hands-on experience in C socket programming, successfully building a TCP port scanner capable of connecting to specified ports and identifying open ones using socket APIs, while utilizing setsockopt() to configure timeouts for improved efficiency. I honed debugging skills by analyzing stack memory and function addresses with gdb, mastering program execution tracing and memory manipulation. Additionally, I learned to implement robust command-line argument validation (e.g., for IPv4 addresses and port ranges) to ensure program reliability and security. By using a Makefile to automate compilation and maintaining a clear directory structure, I developed skills in project management and build automation. This project allowed me to apply cybersecurity concepts in practice, understand the role of port scanning in network reconnaissance and its ethical implications, and cultivate critical thinking and problem-solving abilities by analyzing vulnerable versus secure code and troubleshooting issues to enhance program robustness.

---

9. Ethical Considerations

This project is intended solely for educational and academic purposes. Unauthorized port scanning or exploitation on public or private networks without consent is strictly prohibited and may constitute a legal offense.

---

10. Conclusion

This project, through the development of a buffer overflow demonstration and a TCP port scanning tool, provides a robust foundation for understanding core cybersecurity domains, significantly enhancing security-conscious software engineering skills. Implementing vuln.c and safe.c clearly contrasts the insecurity of gets() with the bounds-checking benefits of fgets(), underscoring the critical role of input validation in preventing memory vulnerabilities. The scanner.c tool, built with socket programming and timeout handling, enables port range scanning, highlighting the practical application and ethical considerations of network reconnaissance. Using gdb to debug stack memory and a Makefile to automate compilation further strengthened debugging and project management skills. Despite limitations like the Makefile lacking a clean target and the scanner's focus on TCP connections, the project offers valuable practice in C programming and secure coding. Future enhancements could include support for SYN or UDP scanning and exploration of advanced exploit techniques, further deepening cybersecurity knowledge and the ability to develop robust software.